Traduction de jeux à information incertaine en réseaux de contraintes stochastiques

Traduction de jeux à information incertaine en réseaux de contraintes stochastiques

Frédéric Koriche, Sylvain Lagrue, Éric Piette, Sébastien Tabary
Université Lille-Nord de France CRIL - CNRS UMR 8188 Artois, F-62307 Lens
{koriche,lagrue,epiette,tabary}@cril.fr

Résumé

Cet article propose d'utiliser le paradigme de la programmation par contraintes stochastiques pour modéliser et identifier des politiques optimales dans les jeux à information incertaine. Nous présentons une traduction permettant de modéliser les jeux décrits dans le formalisme GDL (Game Description Language) en instances du problème d'optimisation de contraintes stochastiques (SCSP). Notre traduction est démontrée correcte pour la classe GDL des jeux à information complète et environnement « indifférent ». L'intérêt de notre approche est illustré par une première résolution d'un jeu GDL en utilisant un solveur SCSP générique.

1 Introduction

La capacité pour un programme de jouer efficacement à n'importe quel jeu de stratégie (GGP pour *General Game Playing*) constitue l'un des défis majeurs de l'IA. Une compétition sur ce sujet est d'ailleurs organisée annuellement par AAAI [8]. Cette capacité générique de jouer a amené les chercheurs à confronter diverses approches efficaces, parmi lesquelles on peut citer des programmes utilisant des méthodes de type Monte Carlo [7], la construction automatique de fonctions d'évaluation [5], ou encore la programmation logique [16] et ASP [13]. Bien entendu, le problème GGP n'est pas cantonné aux « univers ludiques » : il permet de modéliser des problèmes de prise de décision séquentielle mono ou multi-agents.

Dans le contexte GGP, les jeux sont décrits dans un langage de représentation, appelé GDL pour *Game Description Language* [12]. Ce langage, basé sur la programmation logique, permettait dans sa première version de modéliser n'importe quel jeu à information certaine et complète. Une nouvelle version du langage, GDLII (GDL *with Incomplete Information* [17]), permet quant à elle de traiter des jeux à informations incertaines et/ou incomplètes. Dans ce cadre, et de façon usuelle en théorie des jeux, l'incertitude est modélisée par l'intervention de l'environnement, qui est un joueur comme les autres. Les actions de ce joueur, dénommé `random`, ne sont guidées par aucun objectif, mais juste par une distribution de probabilités entre les actions possibles. Il est ainsi possible par ce biais de modéliser des lancers de dés ou de pièces de monnaie, ou encore des distributions de cartes.

Différents formalismes ont été proposés afin de gérer les problèmes liés aux jeux à l'aide de réseaux de contraintes parmi lesquels on peut citer les QCSP [9], les *strategic constraint satisfaction problems* [3] ou encore les *constraint games* [14]. Mais aucun d'entre eux n'aborde les problèmes liés à l'aléatoire ni ne fait le lien avec GDL. L'objectif de cet article est d'aborder le problème GGP d'un point de vue original, fondé sur les réseaux de contraintes stochastiques (SCSP) [18]. En effet, les SCSP sont des outils puissants permettant non seulement de modéliser et de résoudre des jeux à infor-

mations complètes et certaines via QCSP [2], mais également de modéliser l'intervention du hasard. Nous montrons comment réécrire des jeux décrits à l'aide de GDL en SCSP et, par ce biais, comment la résolution du problème correspondant peut identifier une stratégie qui, en espérance, est optimale.

L'article est organisé de la façon suivante. Le formalisme GDL est introduit en Section 2. Nous présentons en particulier une sémantique originale, basée sur les *jeux stochastiques* (ou *jeux de Markov*) [15]. Nous présentons également dans cette section un jeu à information incertaine, le jeu du Verger, qui sert de fil conducteur pour l'ensemble de cet article. Après avoir rappelé en Section 3 le formalisme SCSP, la section 4 propose une traduction (prouvée correcte) d'un fragment de GDL vers SCSP. Avant de conclure, des expérimentations préliminaires utilisant un solveur générique SCSP sont présentées en Section 5.

2 Formalisme GDL

L'objectif de GDL est de fournir un langage générique pour la représentation de n'importe quel jeu, y compris les jeux collaboratifs ou encore les jeux à actions simultanées. La version récente [17] permet de traiter des jeux avec observation partielle où le hasard peut intervenir. Notre étude se focalise sur un fragment particulier de cette version : les jeux à information complète et environnement « indifférent » (*oblivious environment*). Pour ces jeux, les joueurs ont une observation complète de l'état courant. L'environnement est aléatoire mais son comportement ne peut pas être influencé par celui des (autres) joueurs. Ce fragment de GDLII couvre une classe de problèmes très étudiée en théorie des jeux [4], ainsi qu'une variété de jeux de stratégie pour lesquels les joueurs n'ont pas d'effet sur les actions de l'environnement. Un exemple caractéristique est celui des « jeux de dés » où les joueurs peuvent choisir les dés qu'ils lancent, mais ne peuvent pas influencer le comportement des dés.

2.1 Langage

Le langage GDL est dérivé de la programmation logique (avec égalité et négation). Rappelons que l'univers de Herbrand d'un programme logique est défini par l'ensemble des termes instanciés (*grounded terms*) obtenus à partir des symboles de fonctions (incluant les constantes) du programme. Dans un programme GDL, les *joueurs* et *objets* du jeu sont décrits par des constantes, tandis que les *fluents* (ou percepts) et *actions* (coups) sont décrits par des termes. Par exemple, dans le jeu du morpion, le terme $\texttt{cell}(2,2,\texttt{b})$ est un fluent indiquant que la case $(2,2)$ du plateau est marquée par un jeton noir. Nous notons Σ l'univers de Hebrand d'un programme GDL ; Σ_F et Σ_A désignent les sous-ensembles de Σ correspondant respectivement aux termes des fluents et aux termes des actions.

Les atomes d'un programme GDL sont construits à partir d'un ensemble fini de symboles de relations et de symboles de variables. Certains symboles ont un rôle spécifique dans le programme, et sont décrits dans la table 1.

Mot-clé	Description
role(J)	J est un joueur
init(F)	le fluent F fait partie de l'état initial
true(F)	F fait partie de l'état courant
legal(J, A)	J peut faire l'action A
does(J, A)	l'action de J est A
next(F)	F fait partie de l'état suivant
terminal	l'état courant est terminal
goal(J, N)	J reçoit N dans l'état courant
random	le « joueur » environnement

TABLE 1 – Mots-clés GDL

Par exemple, $\texttt{legal}(\texttt{P}, \texttt{mark}(\texttt{X}, \texttt{Y}))$ est un atome indiquant que le joueur P a le droit de marquer la case $(\texttt{X}, \texttt{Y})$ du plateau.

Les règles d'un programme GDL sont des implications composées d'un atome pour le conséquent et d'un ensemble de littéraux (atomes ou négations d'atomes) pour l'antécédent. Par exemple, la règle :

$$\texttt{legal}(\texttt{random}, \texttt{noop}) \leftarrow \texttt{true}(\texttt{control}(\texttt{bob}))$$

signifie que si le joueur bob a actuellement la main alors noop (ne rien faire) est une action légale du joueur random.

Afin d'être *valide*, un programme GDL doit satisfaire plusieurs conditions syntaxiques. Pour des raisons de décidabilité, il doit être *stratifié* [1] et *permis* [11] ; ces deux conditions ne sont pas détaillées pour des raisons de place, mais sont expliquées dans [12]. D'autre part,

comme le but d'un programme GDL est de représenter un modèle formel de jeu, il doit aussi respecter des conditions sur les mots-clés :

(i) `role` apparaît uniquement dans les faits ;

(ii) `true` apparaît uniquement dans le corps des règles ;

(iii) `init` apparaît uniquement dans la tête des règles et ne peut dépendre des mots-clés `true`, `legal`, `does`, `next`, `terminal` et `goal` ;

(iv) `does` apparaît uniquement dans le corps des règles et ne dépend pas de `legal`, `terminal` et `goal` ;

(v) `next` apparaît uniquement dans la tête des règles.

Dans le cadre des jeux à information complète et environnement indifférent, nous ajoutons les conditions :

(vi) Pour chaque fluent `F`, il existe une instance `f` de `F` tel que `init(f)` est un fait.

(vii) Les atomes `legal(random,A)` où `A` désigne une action apparaissent uniquement dans les faits.

2.2 Sémantique

Un modèle courant pour représenter les jeux à information incertaine est celui des *jeux stochastiques* (ou *jeu de Markov*) [15]. Un jeu de Markov à n joueurs sur Σ est un tuple $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R}\rangle$ tel que : [1]

— $N = \{1, \ldots, n\}$ est l'ensemble des *joueurs*,

— S est un ensemble d'*états* ;

— $\boldsymbol{A} = A_1 \times ... \times A_n$, où A_i est un ensemble fini d'actions accessibles au joueur i ;

— $P : S \times \boldsymbol{A} \times S \to [0, 1]$ est la fonction de transition ; $P(s, \boldsymbol{a}, s')$ est la probabilité de passer de l'état s en l'état s' en appliquant l'action jointe $\boldsymbol{a}$;

— $\boldsymbol{R} = \langle r_1, ..., r_n\rangle$, où $r_i : S \to \mathbb{R}$ est la fonction de gain du joueur i.

À ce tuple s'ajoutent l'état *initial* $s_0 \in S$ et l'ensemble des états *terminaux* $S_{ter} \subseteq S$.

Montrons maintenant comment construire un tel modèle à partir d'un programme valide GDL, noté `P`. Un état du jeu est un sous-ensemble de son univers de Herbrand Σ. Puisque les restrictions syntaxiques de GDL garantissent une dérivabilité finie des termes instanciés, tout état est une partie *finie* de Σ_F. Les n instances de `role(R)` définissent les n joueurs du modèle stochastique. Désignons par $n+1$ l'environnement (`random`).

L'état initial est construit à partir des termes de `init(F)`.

Afin de capturer les coups légaux d'un joueur dans un état donné $s = \{f_1, \cdots, f_m\}$, notons $s_{\texttt{true}}$ l'ensemble des faits $\{\texttt{true}(f_1), ..., \texttt{true}(f_m)\}$. Les termes instanciés de `legal(J,A)` dérivables de $\texttt{P} \cup s_{\texttt{true}}$, définissent toutes les actions légales des joueurs `J` dans l'état s. Les états terminaux $\mathcal{S}_{ter}$ et le vecteur de fonctions de gain $\boldsymbol{R}$ sont construits de manière analogue, en utilisant les atomes `terminal` et `goal(R,N)`. Dans un jeu à environnement indifférent, l'ensemble des coups légaux de $n+1$, noté $L(n+1)$, est indépendant de l'état courant : il est construit à partir des termes de `legal(random,A)`.

Pour spécifier la fonction de transition, nous avons besoin de l'état courant ainsi que des actions réalisées simultanément dans cet état. Pour un vecteur $\boldsymbol{a} = \langle a_1, \cdots, a_n, a_{n+1}\rangle$ d'actions jouées par les n joueurs et l'environnement $(n+1)$, notons $\boldsymbol{a}_{\texttt{does}}$ l'ensemble des faits : $\{\texttt{does}(1, a_1), ..., \texttt{does}(n+1, a_{n+1})\}$. L'état suivant est construit à partir des termes instanciés de `next(F)` dérivables de $\texttt{P} \cup s_{\texttt{true}} \cup \boldsymbol{a}_{\texttt{does}}$. De manière concise, cet état suivant est noté $Q(s, \boldsymbol{a})$. En particulier, si $\boldsymbol{a}$ est la concaténation $\boldsymbol{a}' \cdot a_{n+1}$ d'une action jointe $\boldsymbol{a}'$ des n joueurs et d'une action a_{n+1} de l'environnement, alors $Q(s, \boldsymbol{a}' \cdot a_{n+1})$ dénote l'état suivant obtenu à partir de s lorsque, simultanément, les joueurs ont accompli $\boldsymbol{a}'$ et l'environnement à joué a_{n+1}.

La distribution de probabilités sur les transitions est capturée par le comportement stochastique de l'environnement : elle est définie par la distribution uniforme sur tous les coups légaux que peut jouer $n+1$ dans l'état courant. Notons que les états résultants ne sont pas forcément équiprobables. [2]

1. Pour un ensemble U, nous notons 2^U l'ensemble de toutes les parties *finies* de U.

2. Par exemple, un dé « pipé » avec une probabilité de 1/2 de tomber sur 6 peut être modélisé par dix actions légales de `random`, dont cinq donnent le même effet (tomber sur un 6).

Définition 1 *La sémantique d'un jeu valide* GDL P *à environnement indifférent peut être décrite par le jeu stochastique* $\langle N, S, \boldsymbol{A}, P, \boldsymbol{R}\rangle$ *tel que :*

— $N = \{i : \mathtt{P} \models \mathtt{role(i)}$ *et* $\mathtt{i} \neq \mathtt{random}\}$ *;*

— $S = 2^{\Sigma_F}$ *;*

— $A_i = \{a : \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{legal(i,a)}, s \in S\}$ *;*

— $P(s, \boldsymbol{a}, s') = \frac{|\{a_{n+1} \in L(n+1) : s' = Q(s, \boldsymbol{a} \cdot a_{n+1})\}|}{|L(n+1)|}$

— $r_i(s) = \begin{cases} c & si\ \mathtt{P} \cup s_{\mathtt{true}} \models \mathtt{goal(i,c)}, \\ 0 & sinon. \end{cases}$

Naturellement, l'état initial et les états terminaux sont définis par :

$$s_0 = \{f \in \Sigma_F : G \models \mathtt{init}(f)\}$$
$$S_{ter} = \{s \in S : G \cup s_{\mathtt{true}} \models \mathtt{terminal}\}$$

2.3 Un exemple : le Jeu du Verger

Le jeu du *Verger* (*Obstgarten*) est un jeu de société coopératif de 1 à 4 joueurs contre l'environnement. Il se compose de 4 arbres contenant chacun 10 fruits et d'un corbeau composé de 9 pièces. A chaque tour, chaque joueur lance un dé composé de 6 faces :

— 4 faces correspondant chacune à un arbre : le joueur doit alors retirer un fruit de cet arbre ;

— une face corbeau : le joueur doit retirer une pièce au corbeau ;

— une face panier : le joueur doit retirer deux fruits de son choix.

L'objectif est de retirer l'ensemble des fruits de chaque arbre avant que le corbeau ne soit entièrement décomposé. Il est intéressant de noter que la seule et unique décision du joueur intervient lors du tirage du « panier ». La stratégie optimale dans ce cas étant de toujours prendre les fruits dans l'arbre le plus chargé (ou les arbres les plus chargés). Des simulations montrent que cette stratégie permet de gagner dans environ 68,4% des cas, tandis que la pire stratégie (toujours prendre dans l'arbre le moins chargé) ne permet de gagner que dans 53,2% des cas.

Dans le but d'utiliser un exemple court, la figure 1 présente un codage GDL du *Verger* avec un seul joueur (bob), 2 arbres contenant 2 fruits et un corbeau de taille 1. Nous utiliserons pour ce jeu un dé à quatre faces : r (prendre un fruit dans l'arbre rouge), v (prendre un fruit dans l'arbre vert), p (laisser la main au joueur pour qu'il choisisse deux fruits) et c (retirer une pièce au corbeau).

```
% les rôles
role(bob)
role(random)

% opérations sur les entiers
succ(0,0)
succ(0,1)
succ(1,2)

% couleurs des arbres
tree(r)
tree(v)

% initialisation de l'état du jeu
init(state(2,2,1))
init(control(random))
legal(random,roll(D))

% définition des coups légaux du joueur
legal(bob,noop) ← true(control(random))
legal(bob,choice(C1,C2)) ← tree(C1),
                  tree(C2), true(control(bob))

% évolution du jeu
next(control(bob)) ← does(random,roll(p))
next(control(random)) ← true(control(bob))
next(control(random)) ← does(random,roll(R)),
                                      R ≠ p

next(state(R,V,C)) ← true(state(R1,V,C)),
                succ(R,R1),does(random,roll(r))
...
next(state(R,V,C)) ← true(state(R1,V1,C)),
  succ(R,R1), succ(V,V1), does(bob,choice(v,r))

% objectifs et récompenses
goal(bob,100) ← true(state(0,0,C))

% fin du jeu
terminal ← true(state(R,V,0))
terminal ← true(state(0,0,C))
```

FIGURE 1 – Programme GDL du Verger allégé

3 CSP Stochastiques

Le modèle des CSP stochastiques que nous présentons dans cette étude, étend le cadre original de Walsh [18] aux contraintes valuées. De manière formelle, un *Stochastic Constraint Satisfaction Problem* (SCSP) est un 6-tuple $\langle X, Y, D, P, \mathcal{C}, \theta\rangle$ où X représente un ensemble de n variables, Y est un sous-ensemble de X représentant les variables stochastiques, D représente les domaines associés aux variables de X, P est l'ensemble des distributions de probabilités appliquées aux domaines des variables

stochastiques, $\mathcal{C}$ est l'ensemble des contraintes et θ représente une valeur de seuil dans $\mathbb{R}$.

Un SCSP est donc composé de variables de décisions et de variables stochastiques. Les variables de décisions ont le même sens que celles définies dans le cadre des CSP classiques. Pour chaque variable stochastique, on associe une distribution de probabilité aux valeurs des domaines de ces variables. Pour une variable x, on note $D(x)$ le domaine associé à x, et plus généralement, pour un sous-ensemble $Z = \{z_1, \cdots, z_k\}$ de variables dans X, on note $D(Z)$ l'ensemble des tuples de valeurs $D(z_1) \times \cdots \times D(z_k)$.

Chaque contrainte du SCSP est une paire $C = \langle scp_C, val_C \rangle$, où scp_C est un sous-ensemble de X, appelée *portée* de C, et val_C est une fonction qui associe à chaque tuple $\boldsymbol{\tau} \in D(scp_C)$ une valeur (ou utilité) $val_C(\boldsymbol{\tau})$ dans $\mathbb{R} \cup \{-\infty\}$. Une contrainte C est *dure* si son ensemble d'arrivée est $\{-\infty, 0\}$. Dans ce cas, C peut être représentée de manière habituelle par une relation, notée rel_C indiquant les tuples de valeurs autorisées (i.e. les tuples $\boldsymbol{\tau}$ pour lesquels $val_C(\boldsymbol{\tau}) \neq -\infty$). Dans un SCSP les contraintes dures portent uniquement sur des variables de décision. Les autres contraintes, dites *souples*, peuvent porter sur des variables arbitraires.

Une *interprétation* I est un tuple dans $D(X)$ qui associe donc à chaque variable $x \in X$ une valeur dans son domaine $D(x)$. Si l'on note $I_{|Z}$ la projection de I sur le sous-ensemble $Z \subseteq X$, alors l'utilité d'une interprétation I est :

$$val(I) = \sum_{C \in \mathcal{C}} val_C(I_{|scp_C})$$

Une *politique* π est représentée par un arbre dont chaque nœud décrit une variable de l'ensemble X. Les nœuds représentant des variables de décisions ne comportent qu'un seul fils (correspondant à la valeur assignée à cette variable) tandis que les nœuds représentant des variables stochastiques possèdent un fils pour chaque valeur possible du domaine. Les arêtes sont étiquetées par la valeur assignée à la variable correspondante. Les feuilles sont étiquetées par l'utilité de l'assignation associée au chemin depuis la feuille jusqu'à la racine. L'utilité espérée d'une politique π se définie alors comme la somme des utilités des feuilles pondérées par leurs probabilités. Une *solution* du SCSP est une politique π dont l'utilité espérée est supérieure ou égale au seuil θ.

Il est facile de voir que si une politique π contient au moins une branche dont l'assignation viole une contrainte dure, son utilité espérée ($-\infty$) ne peut pas satisfaire le SCSP. Ainsi une politique π satisfait le problème de contraintes stochastiques si elle est « faisable » (i.e. ne viole aucune contrainte dure) et son utilité espérée dépasse le seuil θ.

Considérons par exemple le SCSP de la figure 2. Dans la politique π qui est illustrée, les variables x et z sont décisionnelles (assignées à la valeur 0). La variable y est stochastique, avec trois valeurs (0, 1 et 2) équiprobables. Comme π satisfait systématiquement la contrainte dure C_h, et satisfait la contrainte souple C_s avec une utilité espérée de $2/3 \geq 1/2$, cette politique constitue donc une solution du SCSP.

$X = \{x, y, z\}$	$D(x) = D(y) = D(z) = \{0, 1, 2\}$
$Y = \{y\}$	$P(0) = P(1) = P(2) = 1/3$
$\mathcal{C} = \{C_h, C_s\}$	$C_h : x = z$
	$C_s(x, y) = \begin{cases} 1 & \text{si } x + y \geq 1 \\ 0 & \text{sinon} \end{cases}$
$\theta = 1/2$	

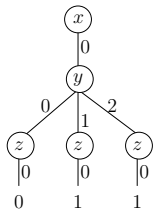

FIGURE 2 – Un SCSP et une politique π

4 De GDL à SCSP

Afin d'obtenir un modèle SCSP à partir d'un programme GDL, il est nécessaire de fixer au préalable un horizon de temps T, dont les pas $t \in \{1, \cdots, T\}$ capturent les tours de jeu. La traduction que nous proposons permet de construire à partir d'un programme GDL P et d'un horizon T, les ensembles de variables X, de domaines D et de contraintes $\mathcal{C}$, ainsi que les distributions de probabilité P du SCSP correspondant. Cette traduction se décompose en quatre étapes que nous allons examiner.

4.1 Élimination des fonctions

Les composants du SCSP associés à un programme GDL P sont extraits à partir des termes, atomes et règles de P. En particulier, les domaines et contraintes font appel aux termes instanciés dérivables de P. Même si avec les restrictions syntaxiques de GDL, il est possible d'utiliser l'approche de la Section 2 pour dériver les termes instanciés de P apparaissant comme fluents et actions, le coût de telles requêtes sur un programme logique stratifié est généralement prohibitif [6].

Afin de circonvenir à ce problème, notre traduction utilise une transformation de P en un programme $\mathtt{P}'$ sans fonction. En utilisant la méthode de Lifschitz et Yang [10], cette transformation se déroule en deux phases : un aplatissement (*flattening*) qui réduit les imbrications de termes, suivi d'une élimination qui remplace les fonctions par des prédicats.

L'aplatissement procède comme suit : pour chaque symbole de fonction f apparaissant dans P, on construit un programme équivalent $\mathtt{P}_\mathtt{f}$ dit f-*aplati* dans lequel toute occurrence d'un terme de la forme $\mathtt{f}(\mathtt{t}_1, \cdots, \mathtt{t}_k)$ est transformée par une égalité de la forme $\mathtt{f}(\mathtt{t}_1, \cdots, \mathtt{t}_k) = \mathtt{Z}$, où Z est une variable de renommage. Par exemple, si P contient l'atome $\mathtt{legal}(\mathtt{random}, \mathtt{roll}(\mathtt{c}))$, alors dans $\mathtt{P}_\mathtt{f}$ l'atome est transformé en la conjonction $\mathtt{legal}(\mathtt{random}, \mathtt{Z}), \mathtt{roll}(\mathtt{c}) = \mathtt{Z}$. En appliquant le f-aplatissement sur toutes les fonctions f de P, le programme obtenu ne contient plus d'imbrication. Notons que l'aplatissement est polynomialement borné par le nombre de fonctions et la profondeur des termes.

La phase d'élimination s'effectue de la manière suivante : chaque symbole de fonction f d'arité k est associé à un symbole de relation F d'arité $k+1$; toute égalité de la forme $\mathtt{f}(\mathtt{t}_1, \cdots, \mathtt{t}_k) = \mathtt{z}$ est remplacée par l'atome $\mathtt{F}(\mathtt{t}_1, \cdots, \mathtt{t}_k, \mathtt{z})$; enfin, la règle [3] :

$$(\exists!\mathtt{z})\mathtt{F}(\mathtt{t}_1, \cdots, \mathtt{t}_k, \mathtt{z})$$

est ajoutée à $\mathtt{P}'$ garantissant ainsi l'équivalence des modèles stables entre P et $\mathtt{P}'$.

3. En programmation logique $\exists!$ signifie "il existe un et un seul".

4.2 Extraction des variables

L'ensemble X des variables du SCSP est obtenu de la manière suivante. On associe à $t \in \{1, \cdots, T\}$, les variables $role_t$, $control_t$, et $terminal_t$. Intuitivement, $role_t$ indique le joueur représenté au tour t, $control_t$ indique quel joueur à la main au tour t, et $terminal_t$ indique si au temps t le jeu est terminé.

Pour chaque instance i de role(J) et chaque $t \in \{1, \cdots, T-1\}$ est engendrée une variable $a_{i,t}$ indiquant l'action du joueur i au tour t. De même, une variable $a_{random,t}$ est construite pour l'action de l'environnement.

Les variables SCSP associées aux fluents sont extraites à partir des prédicats décrivant les fluents du programme $\mathtt{P}'$. Spécifiquement, si $\mathtt{F}(\mathtt{X}_1, \cdots, \mathtt{X}_k, \mathtt{Z})$ est un atome d'arité $k+1$, tel que la variable de renommage apparaît dans un des prédicats init, true et next, alors F est un fluent et la variable f_t est ajoutée à X.

Enfin, comme un programme GDL peut inclure des symboles de relation dits « statiques », qui ne varient pas d'état en état, mais sont utilisés pour établir des relations entre fluents (e.g. succ(I,J)). Pour chaque symbole de relation statique et instant, une variable SCSP est construite. Cependant, comme ces variables sont universelles sur toutes les contraintes où elles apparaissent, on pourra alors dans une phase de prétraitement les retirer de la portée des contraintes.

4.3 Extraction des domaines

Concernant les variables associées au déroulement du jeu, $terminal_t$ est booléenne, et le domaine de $role_t$ et $control_t$ est l'ensemble constitué par les N joueurs (extraits préalablement) et l'environnement (random).

Le domaine de chaque variable fluent f est donné par l'ensemble des combinaisons des constantes $\mathtt{c}_1, \cdots, \mathtt{c}_k$ qui peuvent être instanciées des atomes A de la forme $\mathtt{F}(\mathtt{t}_1, \cdots, \mathtt{t}_k, \mathtt{z})$. Ces domaines sont extraits par filtrage sur le réseau suivant. Pour chaque fluent F d'arité $k+1$ nous construisons les variables $f_1^v, \cdots, f_k^v$ et $f_1^c, \cdots, f_k^c$. Les domaines de $f_1^v, \cdots, f_k^v$ sont initialement formés par tous les symboles de constantes du programme P et le domaine de chaque f_i^c est formé par l'ensemble des symboles de constantes apparaissant en position i

GDL	Variable SCSP	Domaine SCSP
`role(J)`	$role_t$	{`random`, `bob`, `undefined`}
`legal(bob,A)`	$a_{bob,t}$	{{`choice`} × {`r`, `v`} × {`r`, `v`}} ∪ {`noop`, `undefined`}}
`legal(random,A)`	$a_{random,t}$	{{`roll`} × {`c`, `p`, `r`, `v`}}
`next(state(...))`	$state_t$	{(3, 3, 2), ..., (0, 0, 0)} ∪ {`undefined`}
`next(control(...))`	$control_t$	{`bob`, `random`, `undefined`}
`succ(...)`	$succ_t$	{(0, 0), (0, 1), (1, 2), (2, 3), `undefined`}
`tree(...)`	$tree_t$	{`r`, `v`, `undefined`}
`terminal(...)`	$terminal_t$	{`true`, `false`}

TABLE 2 – Variables et domaines au temps t associés au jeu du *Verger*

dans les atomes préfixés par `F` de `P`. Une arête (f_i^v, f_i^c) est construite pour chaque index i, et une arête (f_i^v, g_j^v) est construite dès lors que, dans une règle de `P`, le même symbole de variable apparaît en position i dans un atome préfixé par `F` et en position j par un atome préfixé par `G`. Par arc-consistance, on élimine dans chaque variable f_i^v les constantes ne possédant pas de support. Ainsi, les valeurs de f_i^v sont données par l'union des valeurs de f_i^c et celles des variables voisines g_j^v possédant un support. Enfin, pour chaque instant t, le domaine de la variable fluent f_t est donné par $D(f_t) = D(f_1^v) \times \cdots \times D(f_k^v)$.

L'extraction des domaines des variables d'action s'effectue de manière analogue, en identifiant au préalable, à partir de la relation `legal(j,A)`, les atomes `A` qui participent à la construction du domaine de $a_{j,t}$.

Un point technique : on rajoute la valeur `undefined` à chaque domaine des variables de décisions car si $terminal_t$ prend la valeur `true`, toutes ces variables différentes de *terminal* à un instant supérieur à t sont instanciées à `undefined`.

La figure 2 illustre l'extraction des variables et des domaines obtenues sur le jeu du *Verger* présenté dans la section 2.3.

4.4 Extraction des contraintes

Pour chaque règle `R` du programme GDL `P` et chaque instant t, on associe une contrainte $C_{R,t}$. Les règles de réécritures de la table 3 spécifient la portée des contraintes.

Dans le SCSP, toutes les contraintes sont dures, exceptée la contrainte de score. Ainsi, pour chaque règle `R` du programme `P`, la sémantique de $C_{R,t}$ est une « relation » sur le domaine de sa portée. Cette relation est construite comme suit. Après élimination de

Atome GDL	Portée de la contrainte
`init``(f(...))`	$\{f_0\} \in scp_{C_0}$
`true``(f(...))`	$\{f_t\} \in scp_{C_t}$
`does``(j,a(...))`	$\{a_{j,t}\} \in scp_{C_t}$
`next``(f(...))`	$\{f_{t+1}\} \in scp_{C_t}$
`legal``(j,a(...))`	$\{a_{j,t}\} \in scp_{C_t}$
`goal``(j,N)`	$\{role_t\} \in scp_{score_t}$
`terminal`	$\{terminal_t\} \in scp_{terminal}$

TABLE 3 – règles de réécritures de la portée des contraintes

fonctions, nous savons que tous les termes de `R` dans `P` ont été transformés en prédicats dans la règle correspondante `R'` de `P'`. Supposons que cette règle résultante `R'` soit de la forme :

$$\mathtt{A_1}, \cdots, \mathtt{A_k} \leftarrow \mathtt{B_1}, \cdots, \mathtt{B_{k'}}$$

Notons qu'après élimination des fonctions, l'antécédent peut contenir plusieurs atomes. Comme `R'` exprime en général une implication, la relation de $C_{R,t}$ doit capturer cette implication. Dans ce but, soit `A` (resp. `B`) l'ensemble $\{\mathtt{A_1}, \cdots, \mathtt{A_k}\}$ (resp. $\{\mathtt{B_1}, \cdots, \mathtt{B_{k'}}\}$). Soit $\mathtt{U(A_i)}$ l'ensemble des combinaisons de constantes qui sont instances de $\mathtt{A_i}$, et `U(A)` (resp. `U(B)`) la jointure de ces combinaisons sur `A` (resp. `B`). D'autre part, soient `C(B)` (resp. `C(AB)`) l'ensemble des combinaisons de constantes qui sont instances de la conjonction sur `B` (resp. $\mathtt{A} \cup \mathtt{B}$). L'ensemble des combinaisons de constantes qui sont instances de `R'` est donc :

$$\mathtt{C(AB)} \cup (\mathtt{U(A)} \bowtie (\mathtt{U(B)} \setminus \mathtt{C(B)}))$$

Cet ensemble d'instances peut être obtenu par application de requêtes conjonctives. La projection de cet ensemble sur $scp_{C_{R,t}}$ donne $rel_{C_{R,t}}$.

Notons que par la règle d'information complète (vi), les contraintes associées à `init` n'autorisent qu'une seule valeur par variable de fluent f_0. La contrainte souple $score_t$ associe

à chaque joueur j la valeur N donnée dans **goal**(j,N) si le corps de la règle associée à cet atome est vrai, et la valeur 0 sinon.

Pour terminer, la composante P du SCSP est formée en associant une distribution de probabilité uniforme sur le domaine de $random_t$ à chaque instant t.

4.5 Équivalence des modèles

Dans le modèle SCSP obtenu, notons $X_{f,0}$ l'ensemble des variables de fluent apparaissant dans l'état initial, et $X_{f,t}$ (resp. $X_{a,t}$) l'ensemble des variables de fluent (resp. d'action) au temps t. Par construction,

$$X = \left(\bigcup_{t=0}^{T} X_{f,t}\right) \cup \left(\bigcup_{t=1}^{T-1} X_{a,t}\right)$$

La réécriture obtenue détermine un graphe orienté sans circuit (DAG) étiqueté $G_{\text{SCSP},T}$ dont l'ensemble des sommets S et des arcs A est le plus petit ensemble construit de la manière suivante :

— l'unique tuple $s_0 \in D(X_{f,0})$ qui est autorisé par le SCSP est un sommet de S ;

— si $s_t \in D(X_{f,t})$ est un sommet de S et $s_{t+1} \in D(X_{f,t+1})$ tel que la concaténation $s_t \cdot s_{t+1}$ est un tuple autorisé par le SCSP, alors s_{t+1} est un sommet de S et (s_t, s_{t+1}) est un arc de A.

Chaque arc $(\boldsymbol{s}_t, \boldsymbol{s}_{t+1})$ est étiqueté par la probabilité $\frac{k}{d}$ où d est la taille de $D(a_{\texttt{random},t})$ et k est le nombre de tuples $(\boldsymbol{s}_t, \texttt{a}, \boldsymbol{s}_{t+1})$ autorisés pour lesquels $\texttt{a} \in D(a_{\texttt{random},t})$. Enfin, chaque sommet s_t est étiqueté par une table associant à chaque joueur i dans le domaine de $role_t$ son utilité donnée par $score_t(i)$.

De manière analogue, nous associons à un programme GDL, noté P, le DAG suivant : chaque état s du jeu de Markov de P est réécrit en éliminant les symboles de fonctions (cf Section 4.1) et en ne conservant que la combinaison de constantes du prédicat ; l'ensemble des sommets est l'ensemble S des états (transformés) du jeu de Markov de P′ atteignables depuis l'état initial par un chemin de longueur au plus $T-1$. [4] Il existe un arc (s_t, s_t+1) entre deux sommets ssi il existe un vecteur d'actions légales $\boldsymbol{a}$ pour s_t, tel que $s_{t+1} = Q(s_t, \boldsymbol{a})$. Chaque arc (s_t, s_t+1) est étiqueté par la proportion d'actions a de **random** pour lesquelles $s_{t+1} = Q(s_t, \boldsymbol{a}\cdot\texttt{a})$. Enfin, chaque sommet s_t est étiqueté par son vecteur de récompenses $\boldsymbol{R}(s_t)$.

Proposition 1 *Pour tout horizon T, si SCSP est la traduction du programme GDL* P *alors les graphes $G_{SCSP,T}$ et $G_{\texttt{P},T}$ sont identiques.*

Preuve (schéma général). De part la règle de réécriture associée à init, le sommet s_0 de $G_{\text{SCSP},T}$ appartient aussi à $G_{\texttt{P},T}$. Par hypothèse d'induction, supposons que s_t appartienne à la fois à $G_{\text{SCSP},T}$ et $G_{\texttt{P},T}$. Considérons un état $s_{t+1} \in D(X_{f,t+1})$ tel que $s_t \cdot s_{t+1}$ est un tuple autorisé par le SCSP. Alors il existe un tuple $\boldsymbol{a} \in D(X_{a,t})$ tel que $s_t \cdot \boldsymbol{a}\boldsymbol{s}_{t+1}$ est autorisé par le SCSP, ce qui implique que $\boldsymbol{a}$ est un vecteur d'actions légales, et donc s_{t+1} appartient à la fois à $G_{\text{SCSP},T}$ et $G_{\texttt{P},T}$. De la même manière, l'arc (s_t, s_{t+1}) appartient aux deux graphes. Comme l'environnement est indifférent et que son domaine d'actions est le même pour P et SCSP, les étiquettes sur les arcs sont identiques. Enfin, par la réécriture de **goal**(j,N) en $score_t$, les étiquettes sur les sommets sont identiques.

5 Résolution et expérimentations préliminaires

Pour simplifier la résolution du réseau de contraintes stochastiques obtenu, nous utilisons des techniques de prétraitement.

La première technique utilisée est la fusion des contraintes de même portée. Ainsi, deux contraintes c_i et c_j (de relation rel_{c_i} et rel_{r_j}) tel que $scp_{c_i} = scp_{c_j}$ deviennent une unique contrainte c_k de relation $rel_{r_k} = rel_{c_i} \cup rel_{c_j}$.

La seconde technique utilisée est la suppression de toutes les contraintes unaires (d'arité 1). Le domaine des variables présentes dans ces contraintes est restreint aux valeurs satisfaisant les tuples de la relation associée.

Une dernière technique consiste en la détection des variables universelles dans chaque contrainte. Quelque soit la valeur prise par ces variables, la contrainte est toujours satisfaite.

4. La longueur du chemin est donnée par le nombre de ses arcs.

Ces variables sont alors supprimées de la portée des contraintes. Une variable x est universelle dans une contrainte c_i si le nombre de tuples de rel_{c_i} est égal au produit de la taille du domaine de x avec le nombre de tuples de la relation associée à la contrainte c_j telle que $scp_{c_i} \setminus \{x\} = scp_{c_j}$.

Algorithme 1 : search

```
   Données : seuil θ_h politique π, entier i
   Résultat : politique π, satisfaction θ
1  si i > |X| alors
2      retourner (π, 1)
3  θ ← 0
4  pour chaque v ∈ D(x_i) faire
5      si isConsistent(x_i, v, C) alors
6          si x_i ∈ S alors
7              p ← prob(x_i, v)
8              (π, θ_t) ← search((θ_h − θ)/p, π, i + 1)
9              θ ← θ + p ∗ θ_t
10             si θ ≥ θ_h alors
11                 retourner (π, θ)
12         sinon
13             π ← sol ∪ {(x_i, v)}
14             (π, θ_t) ← search(θ_h, π, i + 1)
15             θ ← max(θ, θ_t)
16             si θ ≥ θ_h alors
17                 retourner (π, θ)
18             sinon
19                 π ← π \ {(x_i, v)}
20 retourner (π, θ)
```

Le solveur SCSP utilisé pour résoudre le réseau de contraintes stochastiques obtenu utilise une méthode de recherche arborescente avec retour arrière sur l'ensemble des valeurs du domaine de chaque variable. L'algorithme 1 illustre cette résolution. La méthode est appelée avec le seuil souhaité, une politique π initialement vide et l'indice de la première variable à assigner (initialement d'indice 0). Elle retourne une politique π qui satisfait les contraintes dures et tel que la probabilité de satisfaire les autres contraintes est supérieure ou égale au seuil θ_h. Notons que si le SCSP est insatisfiable, la politique retournée est vide. La fonction $isConsistent(x, v, C)$ retourne vrai si la valeur v pour la variable x satisfait l'ensemble des contraintes C. De même, la probabilité d'obtenir la valeur v pour la variable stochastique x est donnée par la fonction $prob(x, v)$.

Le tableau 4 indique les résultats obtenus sur le jeu du Verger allégé (deux fruits par arbre et un corbeau de taille 1) avec différents horizons et un seuil de 50%.

Horizon	Temps	$\#\pi$
1	29 ms	0
2	28 s	0
3	4 min 53 s	0
4	37 min	6
5	4 h 54 min	12

TABLE 4 – Politiques gagnantes du Verger 2 2 1 avec $\theta \geq 50\%$

Les valeurs des variables $a_{bob,t}$ et $state_{0,t}$ à chaque temps dans les politiques satisfaisant le réseau de contraintes obtenu pour le jeu du Verger d'horizon 4 avec un seuil de 50% sont répertoriées dans le tableau 5. Les valeurs de $state_{0,t}$ sont des triplets représentants le nombre de fruits dans les arbres r et v et la taille du corbeau c. Les valeurs de la variable $a_{bob,t}$ correspondent à l'action « ne rien faire » (noop) et aux quatre actions de choix de fruits : (r,v), (v,r), (r,r), (v,v). Le joueur *random* lance le dé avec une probabilité équiprobable. On peut noter que les stratégies gagnantes sont celles faisant intervenir cette dernière action. C'est pourquoi, il est intéressant de noter qu'à $t = 1$ la quantité de fruits dans chaque arbre n'a pas été modifiée. Les stratégies optimales présentées dans le tableau à $t = 1$, nous indique que la stratégie optimale est de sélectionner un fruit dans chaque arbre. Les quatre premières solutions représentent le cas où le joueur *random* tire deux fois de suite le panier. Les solutions 5 et 6, le cas où le joueur a le choix au temps 1 puis le joueur *random* tire un fruit dans l'arbre v puis dans l'arbre r (solution 5) ou l'inverse (solution 6). Notons que le nombre de solutions est conditionné par la valeur de seuil. Avec un seuil de 50%, les solutions retournées correspondent à la stratégie optimale attendue dans le jeu du Verger.

6 Conclusion

Dans cet article, nous avons proposé de modéliser sous la forme de réseaux de contraintes stochastiques des jeux GDL à information incertaine et complète où un joueur représentant l'environnement joue indifféremment des actions des autres joueurs. La traduction de ce fragment de GDL utilise une sémantique basée

Variable	t=1	t=2	t=3	t=4
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice v r	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 1 1	0 0 1
$a_{bob,t}$	choice r v	noop	choice r v	undefined
$state_{0,t}$	2 2 1	1 1 1	1 0 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined
$state_{0,t}$	2 2 1	1 1 1	0 1 1	0 0 1
$a_{bob,t}$	choice v r	noop	noop	undefined

TABLE 5 – Politiques gagnantes du Verger d'horizon 4 (où à l'état initial $state_{0,0}$ = (2 2 1) et $a_{bob,0}$ = noop)

sur un jeu de Markov. Nous avons montré que notre réécriture est correcte par l'équivalence des modèles à chaque horizon. Nos premières expérimentations sur le jeu du Verger à l'aide d'un solveur SCSP générique sont concluantes et permettent de mettre en avant la stratégie optimale pour un horizon donné.

À court terme, nous souhaitons étendre notre modèle à un environnement dont les actions du joueur *random* peuvent dépendre de l'état courant. De même, des méthodes de filtrages adaptées au SCSP peuvent être utilisées afin d'accélérer la recherche de modèles. Plus généralement, ce travail offre de nombreuses perspectives dont notamment l'apprentissage automatique de stratégies optimales.

Références

[1] K. R. Apt, H. A. Blair, and A. Walker. Foundations of deductive databases and logic programming. chapter Towards a Theory of Declarative Knowledge, pages 89–148. Morgan Kaufmann, 1988.

[2] T. Balafoutis and K. Stergiou. Algorithms for stochastic CSPs. In *Proccedings of CP'06*, pages 44–58, 2006.

[3] C. Bessiere and G. Verger. Strategic constraint satisfaction problems. In *Proceedings of CP'06 Workshop on Modelling and Reformulation*, pages 17–29, 2006.

[4] N. Cesa-Bianchi and G. Lugosi. *Prediction, Learning, and Games*. Cambridge, 2006.

[5] J. E. Clune, III. *Heuristic evaluation functions for general game playing*. PhD thesis, University of California, Los Angeles, USA, 2008. Adviser-Korf, Richard E.

[6] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3) :374–425, 2001.

[7] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *Proc. of AAAI'08*, pages 259–264, 2008.

[8] M. Genesereth, N. Love, and B. Pell. General game playing : Overview of the aaai competition. *AAAI Magazine*, 26(2) :62–72, 2005.

[9] I. P. Genta, P. Nightingalea, A. Rowleya, and K. Stergiou. Solving quantified constraint satisfaction problems. *Artificial Intelligence*, 172(6-7) :738–77, 2008.

[10] V. Lifschitz and F. Yang. Eliminating function symbols from a nonmonotonic causal theory. In *Knowing, Reasoning, and Acting : Essays in Honour of Hector J. Levesque*. College Publicaions, 2011.

[11] I. W. Lloyd and R. W. Topor. A basis for deductive database systems. ii. *J. Log. Program.*, 30(1) :55–67, Apr. 1986.

[12] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing : Game description language specification. Technical report, 2008.

[13] M. Möller, M. T. Schneider, M. Wegner, and T. Schaub. Centurio, a general game player : Parallel, Java- and ASP-based. *Künstliche Intelligenz*, 25(1) :17–24, 2011.

[14] T.-V.-A. Nguyen, A. Lallouet, and L. Bordeaux. Constraint games : Framework and local search solver. In *Proccedings of International Conference on Tools with Artificial Intelligence (ICTAI'13)*, pages 8–12, 2013.

[15] Y. Shoham and K. Leyton-Brown. *Multiagent Systems : Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

[16] M. Thielscher. Flux : A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5(4-5) :533–565, 2005.

[17] M. Thielscher. A general game description language for incomplete information games. In *Proc. of AAAI'10*, pages 994–999, 2010.

[18] T. Walsh. Stochastic constraint programming. In *Proc. of ECAI'02*, pages 111–115, 2002.